DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Tercer Trabajo Práctico

Junio de 2010 — Algoritmos y Estructuras de Datos III

Integrante	LU	Correo electrónico
Bianchi, Mariano	92/08	marianobianchi08@gmail.com
Brusco, Pablo	527/08	pablo.brusco@gmail.com
Di Pietro, Carlos Augusto Lyon	126/08	cdipietro@dc.uba.ar

Facultad de Ciencias Exactas y Naturales

Universidad de Buenos Aires

Ciudad Universitaria - (Pabellón I/Planta Baja)

Intendente Güiraldes 2160 - C1428EGA

Ciudad Autónoma de Buenos Aires - Rep. Argentina

Tel/Fax: (54 11) 4576-3359

http://www.fcen.uba.ar

Índice

1. Ejercicio 1

1.1. Introducción

El problema a resolver en el presente trabajo práctico consiste en dado un grafo simple, encontrar un *MAX-CLIQUE* para dicho grafo. Una clique es un subgrafo completo del grafo original. Un *MAX-CLIQUE* , es una clique tal que no exista otra que contenga más vértices.

Este problema es muy conocido. Además, no está computacionalmente resuelto y tiene infinidad de aplicaciones en distintos problemas de la vida real, lo que hace que sea un importante objeto de estudio. Algunas de sus aplicaciones más estudiadas provienen de áreas como bioinformática, transporte, diseño de tuberías, diseño de redes energéticas, procesamiento de imágenes, seguridad informática, electrónica, etc.

1.2. Algunas aplicaciones

1.2.1. Telefonía Móvil

Una aplicación posible podría darse por ejemplo, en el contexto de una compañía de telefonía móvil. Como bien se conoce, este tipo de empresa ofrece un plan llamado "plan empresas" para el cuál todos los teléfonos que se encuentren bajo este, tienen la posibilidad de comunicarse entre sí de forma libre.

Para la empresa, podría ser de interés conocer algún grupo de personas que estén comunicados todos entre sí para ofrecerles un "plan empresas" y así beneficiarlos dándole la oportunidad de aumentar el caudal de llamadas entre sí, por un precio más razonable.

Podemos pensar el modelo de la siguiente manera:

- Vértices: Son los celulares de los clientes de la empresa de telefonía.
- Ejes o aristas: Existe una arista entre dos vértices (o teléfonos) A y B si alguna vez se realizó una llamada entre ambos (A llamó a B o vice versa)[1].

Con este modelo, encontrar una clique de tamaño K significa encontrar un grupo de K celulares que se hayan comunicado todos entre sí alguna vez durante un período de tiempo predeterminado. Pasa lo mismo si se busca un *MAX-CLIQUE* . Esto sería equivalente a encontrar el mayor grupo de personas que estén comunicados todos entre sí[2].

De la misma forma, se puede pensar al revés, y quizás a la empresa le interese conocer grupos de celulares que estén todos comunicados entre sí, para NO ofrecerles el "plan empresas" ya que de esa manera ese grupo de personas podría eventualmente bajar el consumo de sus llamadas descendiendo las ganancias de la empresa de telefonía.

[1] Sería conveniente elegir un período de tiempo acotado para ver si se produjo dicha llamada o no y así poder armar el grafo.

[2] Una variante sería buscar el *MAX-CLIQUE* durante un año todos los días y quedarse con aquellos que se haya repetido la mayor cantidad de veces.

1.2.2. Planes Sociales

Imaginemos que el gobierno de la nación quiere lanzar un nuevo plan social y quiere que la entrega de estos sea lo más equitativa posible para la sociedad. En este sentido, quiere entregar planes a la mayor cantidad de personas posibles de forma tal que ninguna persona que reciba el beneficio del plan esté relacionada directamente con otra que también lo reciba. Por relacionada directamente, se entiende que esas personas no tengan un parentesco directo que las una (madre,padre,hijo/a). Podemos pensar el siguiente modelo de grafos:
Los nodos son las personas que pueden verse beneficiadas con el plan (i.e: mayores de 18 años que tengan hijos) y existe un eje que une un par de nodos si existe un parentesco directo que una a las dos personas que representan esos nodos.

Lo que se debería buscar en este modelo entonces es el mayor conjunto de nodos independientes, es decir, un conjunto de nodos tales que para un par cualquiera de esos nodos no exista una arista que los una. Esto no es directamente transferible a un problema de *MAX-CLIQUE* pero lo es si tomamos el complemento del grafo proveniente del modelo anteriormente mencionado. Haciendo esto, sabemos que el grafo resultante tiene ejes donde antes no había y le faltan los ejes que antes existían. Por lo que, si antes había un conjunto de nodos independientes, en el complemento en su lugar hay una clique. Por lo tanto ahora sí podemos ver que encontrar una *MAX-CLIQUE* en el complemento del grafo creado como antes se mencionó es igual a encontrar un grupo de personas no relacionadas entre sí para poder asignarles un plan social.

1.2.3. Buscador Web

Como bien sabemos, en un buscador web se realizan muchísimas búsquedas diarias. Para la empresa que mantiene un buscador web, puede ser de gran importancia saber cuál es el tema o palabra más buscado/a en algún período de tiempo en particular. Podemos pensar por ejemplo el siguiente modelo para las búsquedas realizadas:
Los nodos representan una palabra o frase que haya sido buscada en el sitio web. Las aristas aparecen entre dos nodos si entre esas frases hay alguna palabra en común (o un cierto porcentaje de palabras en común, por ejemplo, para evitar que dos frases estén relacionadas sólo por tener una preposición en común).

Para este modelo, encontrar una clique significa encontrar un conjunto de frases que (en cierto sentido) hacen referencia a una misma temática. Por lo tanto, encontrar una *MAX-CLIQUE* sería equivalente a conocer cuál es el tema (o palabra) más consultado en el buscador web.

2. Ejercicio 2

2.1. Introducción

En esta sección se presentara un algoritmo exacto para resolver el problema de encontrar la Clique Máxima en un grafo.

Aun no se conocen algoritmos buenos, es decir, polinomiales con respecto al tamaño de la entrada, para resolver este problema, asi que nos consentraremos en realizar mejoras al algoritmo de fuerza bruta que considera todos los casos.

2.2. Explicación

Un algoritmo de fuerza bruta para resolver el problema de Max-Clique podría simplemente intentar formar el conjunto más grande de nodos, donde ese conjunto sea completo, intentando todas las posibilidades eligiendo todos los conjuntos de un cierto tamaño, luego intentar con un tamaño menor, etc. Probablemente la complejidad de un algoritmo de este estilo sea n^n donde n es la cantidad de nodos del grafo.

Una mejora que surge casi inmediatamente es utilizar la técnica de BackTracking, cuya función principal es intentar podar el arbol implicito de combinaciones posibles.

De todas maneras, implementar solo un BackTracking parece ser poco con respecto a las mejoras que se pueden lograr. A continuación, se explicara el algoritmo implementado con un pseudocodigo y se verá cada una de las mejoras por separado. **Algoritmo Exacto**(G: grafo)

```
1  CliqueMayor = vacia
2  componentesConexas = DetectarComponentesConexas(G)
3  Para cada componente en componentesConexas {
4      heap = crearHeap(G,componentes)
5      Mientras noVacio(heap) ∧ top(heap) ≥ tam(CliqueMayorActual){
6          v = top(heap)
7          Para tamCliqueABuscar desde grado(v)+1 hasta tam(CliqueMayorActual)+1{
8              vecinosFiltrados = filtrarVecinosMenores(v, tamCliqueABuscar-1)
9              Si tam(vecinosFiltrados) +1 ≥ tamCliqueABuscar
10                 temp = BuscoCliqueDeTamañoK(tamCliqueABuscar, vecinosFiltrados)
11                 Si tam(CliqueMayorActual) < tam(temp)
12                     CliqueMayorActual = temp
13         }
14     }
```

Algoritmo 1: Pseudocódigo del algoritmo exacto

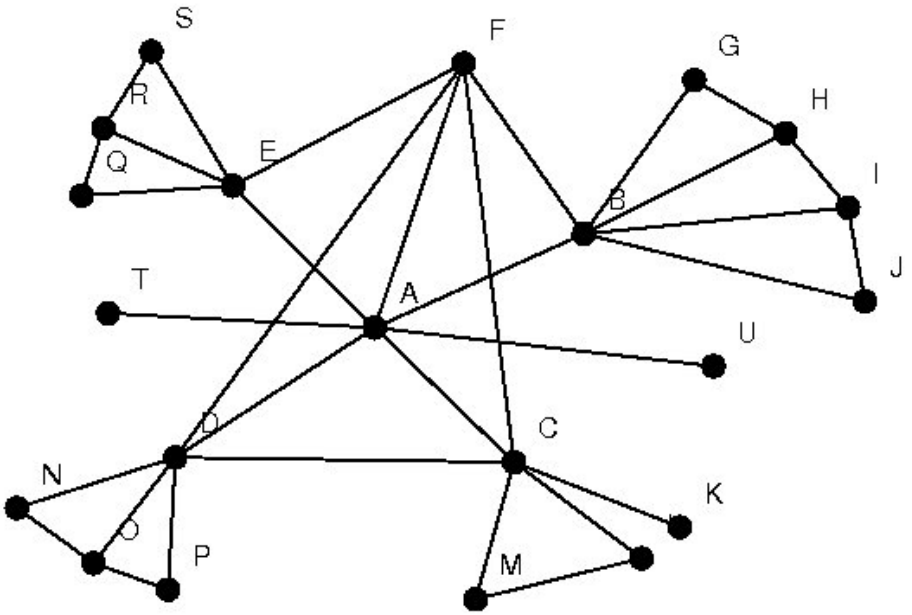
Primero, se detectan las componentes conexas del grafo, ya que buscar la max clique en todo el grafo es equivalente a quedarse con la máxima de las max cliques de cada componente conexa.

Luego, para cada componente, se crea un heap que contiene los nodos de la componente ordenados por mayor grado. Esto no parece tener mucha importancia, pero se aclarará a medida que se avanza con la explicación.

A partir de esta estructura, vamos obteniendo en cada iteración, el nodo v de mayor grado disponible en la componente que no hayamos analizado.

Una vez que tenemos el nodo v, lo que se hace es buscar la mayor clique en la cual está contenido. Para ello utilizamos la función BuscoCliqueDeTamañoK a la cual le indicamos el tamaño de la clique que queremos buscar (desde el grado de $v + 1$) y los vecinos de v que tienen posibilidades de pertenecer a la clique (dado su grado).

Por ejemplo, veamos la siguiente figura:



Como se ve en el ejemplo, si buscamos cliques que contengan al vertice A (cuyo grado es 7), no tendria sentido buscarlas de cualquier tamaño, sino desde el grado del vertice más uno, es decir, cliques de tamaño 8 en este caso. Pero si vemos cuantos nodos vecinos de grado 7 tiene, notamos que no alcanzan para formar una clique de 8. Luego buscamos nodos vecinos de A de grado 6 o más y notamos que hay 3 (B, C, D) y A, por lo tanto tampoco alcanzan para una clique de tamaño 7. Luego los de grado 5 o más, encontramos 5 más A. En este caso, hay probabilidades de que estos 6 nodos puedan formar una clique de tamaño 6, entonces realizaremos una busqueda exaustiva utilizando backtracking, el cual en este caso dara negativo ya que no hay cliques de grado 6 en el grafo que contengan a A. Una vez más,

buscaremos una clique de tamaño 5, para lo cual solo miraremos nodos de grado 4 o más y realizaremos el backtracking sobre los nodos disponibles.

De todas maneras, el algoritmo intentará buscar cliques cada vez mas chicas hasta llegar a cliques de del tamaño de la más grande encontrada hasta el momento sin incluir, ya que encontrar cliques más chicas no tiene importancia.

El heap de nodos ordenados por grados tiene gran importancia, ya que permite decidir rapidamente cuando terminar con la busqueda. Esto se realiza comprobando que el nodo siguiente del heap, es decir, el de mayor grado de la componente, supere o iguale al tamaño de la clique mas grande encontrada, ya que en caso contrario, no tendría sentido seguir buscando.

3. Ejercicio 3

3.1. Introducción

Para la resolución de este ejercicio se debía desarrollar e implementar una heurística constructiva para resolver el problema de encontrar un *MAX-CLIQUE* dado un grafo simple.

3.2. Explicación

En una primera aproximación al problema, se pensó un algoritmo bastante sencillo. La idea del mismo radicaba en ir tomando los nodos en orden de grados, es decir, comenzando con los de mayor grado hasta llegar a los de menor grado. De esta forma, uno puede pensar que al tomar primero los vértices de mayor grado, hay mas chances de encontrar una clique de mayor tamaño.

Esto es claramente una heurística válida que utiliza la técnica de algoritmo goloso. Pero es claro también que se pueden encontrar fácilmente ejemplos de grafos en los que dicho algoritmo funcione tan mal como uno quiera.

A fines de evitar en cierto grado muchos casos para los cuales este algoritmo funciona mal, se planteó uno nuevo que utiliza la misma idea pero que la misma no se realiza sobre todos los nodos del grafo sino que se hace sobre un subconjunto de los mismos. Para formar dicho conjunto, se implementó un algoritmo que revisa todas las combinaciones de 2 vértices distintos (siempre y cuándo haya 2 o más vertices en el grafo) que sean vecinos entre sí y se guarda en un conjunto de vértices aquellos que sean vecinos a ambos vértices y además se guardan los 2 vértices en cuestión. Esto se repite para cada posible combinación de vértices de a 2, guardando siempre el conjunto más grande que se haya encontrado completado de la forma antes mencionada.

Una vez encontrado este subgrafo, se procede a realizar el algoritmo goloso antes mencionado pero sobre dicho subgrafo, es decir, se busca el nodo con mayor grado en ese subgrafo y se coloca en un conjunto, el cuál será devuelto como clique al terminar el algoritmo. Luego, para el resto de los nodos del subgrafo, se va tomando de a uno a la vez en orden de mayor a menor grado (considerando sólo los adyacentes que pertenecen al subgrafo) y se verifica que sea adyacente a todos los que pertenecen a la clique. Si lo es, entonces se lo inserta en el conjunto sino se lo descarta. Finalmente, se prosigue con estos pasos hasta haber intentado con todos los nodos del subgrafo devolviendo entonces la clique encontrada.

A continuación se adjunta el pseudocódigo del algoritmo constructivo antes descripto y el de las funciones auxiliares pertinentes. En los mismos, utilizaremos a “n” como forma de expresar la cantidad de nodos pertenecientes al grafo.

cliqueConstructivo(G: grafo)

```
1  frontera = mayorFronteraEnComun(G) ;                    // O(n^3)
2  res = ∅;                                                // O(1)
3  mientras frontera ≠ ∅
4     v = elMasRelacionado(G,frontera);                    // O(n^2)
5     si esVecinoDeTodos(G,res,v);                         // O(n)
6        insertar v en res;                                // O(log(n))
7     fin si
8     eliminar v de frontera;                              // O(log(n))
9  fin mientras
10 si res == ∅;                                            // O(1)
11    insertar v_0 en res;                                 // O(1)
12 fin si
13 devolver res
```

Algoritmo 2: Pseudocódigo del algoritmo constructivo

mayorFronteraEnComun(G: grafo)

```
1  aux = ∅;                                                // O(1)
2  res = ∅;                                                // O(1)
3  paratodo u,v ∈ V_G tq (u,v) ∈ X_G
4     aux = (adyacentes(G,u) ∩ adyacentes(G,v)) ∪ u ∪ v;   // O(n)
5     si #aux > #res;                                      // O(1)
6        res = aux;                                        // O(1)
7     fin si
8  fin paratodo
9  devolver res
```

Algoritmo 3: Pseudocódigo de un algoritmo secundario al constructivo

El pseudocódigo de las funciones *elMasRelacionado, esVecinoDeTodos* y *adyacentes* no se detalla ya que no son algoritmos de gran complejidad. Igualmente, se explicarán brevemente a continuación.
En el caso de *elMasRelacionado*, recibe dos parámetros, un grafo y un conjunto de vértices. Esta función devuelve el vértice de mayor grado del grafo inducido por los vértices de ese conjunto. La función *esVecinoDeTodos* recibe un grafo, un vértice y un conjunto de vértices. Devuelve verdadero si y sólo si el vértice es adyacente a todos los vértices del conjunto. Por último, en el caso de la función *adyacentes*, recibe un grafo y un vértice como parámetros y devuelve el conjunto de todos los vértices adyacentes al pasado como parámetro.

3.3. Análisis de la complejidad del algoritmo

Para realizar el siguiente análisis de complejidad vamos a remitirnos al pseudocódigo de la función *AlgoritmoConstructivo* adjunto en la sección anterior. Cabe recordar que cada vez que se haga mención a "n" nos estaremos refiriendo a la cantidad de nodos del grafo al que se quiere analizar.
En la primer línea se realiza una llamada a la función *mayorFronteraEnComun* y una asignación. Estas 2 operaciones tienen una complejidad de $O(n^3)$. Más adelante se explicará el por qué de la misma. Luego, dentro del ciclo, la función que mayor complejidad temporal tiene es

elMasRelacionado y la misma es $O(n^2)$. Esta complejidad se debe a que para cada vértice del grafo inducido por el conjunto de vértices pasado como parámetro, se debe calcular cuántos vecinos tiene dentro de ese grafo, para lo que se debe recorrer todo el conjunto una vez por cada vértice. Como a lo sumo puede haber "n" vértices en ese conjunto, debo recorrerlo "n" veces por cada vértice. Entonces, su complejidad es $O(n^2)$.
Deberiamos ver ahora cuántas veces va a ejecutarse el ciclo. Este finalizará una vez que el conjunto llamado "frontera" quede vacío. Este arranca con el valor de la mayor frontera en común, por lo que a lo sumo puede ser de tamaño "n", es decir, todos los vértices pueden pertenecer a él en el peor caso. Podemos observar además, que en cada paso del ciclo, su tamaño disminuye en uno (línea 8, al eliminar un vértice del conjunto) , por lo que a lo sumo el ciclo se ejecutará "n" veces. Como la función más compleja dentro del ciclo era *elMasRelacionado* con una complejidad de $O(n^2)$ y la misma se ejecutará "n" veces, entonces la complejidad total temporal del ciclo es de $O(n^3)$. Por lo tanto, la complejidad del algoritmo constructivo es de $O(n^3)$.

Para completar este análisis, falta justificar por qué la función *mayorFronteraEnComun* tiene complejidad $O(n^3)$.
En el caso de la intersección, sólo hace falta recorrer una sola vez la lista de todos los vértices para saber cuáles son vecinos a ambos nodos a la vez, y la unión de dicha intersección con los vértices "u" y "v" tiene complejidad logarítmica, por lo que dicha instrucción tiene complejidad $O(n)$. Pero esta sentencia se ejecuta tantas veces como se ejecute el ciclo *for*, así que para saber la complejidad total, debemos conocer las veces que se ejecuta el ciclo: éste se ejecuta tantas veces como aristas haya. En el peor caso, si hay "n" nodos entonces puede haber hasta "n^2" aristas. Por lo que el ciclo se ejecutará a lo sumo n^2 veces[3]. Finalmente, la complejidad total de *mayorFronteraEnComun* es $O(n^3)$.

3.4. Detalles de implementación

3.5. Resultados

3.6. Debate

3.7. Conclusiones

[3]La complejidad podría verse como $O(m)$ pero como se eligió una matriz de adyacencia para modelar el grafo, para recorrer todas las aristas es necesario pasar por cada valor de la matriz de adyacencia, por eso es que se toma como complejidad $O(n^2)$ y no $O(m)$

4. Ejercicio 4

4.1. Introducción

En este ejercicio, nos fue requerida una implementación de un algoritmo que utilice la técnica de búsqueda local para resolver el problema de *MAX-CLIQUE* .

4.2. Explicación

La idea general de la búsqueda local es partir de una solución parcial, ya sea una dada por algún algoritmo constructivo o alguna solución trivial y de allí realizar una búsqueda de soluciones vecinas que pueda mejorar la solución parcial encontrada antes.

Para comenzar con el algoritmo, se decidió que la solución de partida sea la que se obtiene a partir del algoritmo constructivo implementado como solución del ejercicio 3, que se detalla en la sección [3].
Luego, se definió qué iba a ser tomado como vecindad, es decir, qué conjunto de soluciones iban a ser tenidas en cuenta a la hora de realizar la búsqueda local. Para definir esta vecindad, hubo que tener en cuenta que la misma debía ser fácil de calcular. Además, el criterio de parada debía ser cuando se encontrara un máximo local.

Como la vecindad a definir debía ser lo suficientemente simple y debía mantener un espacio de búsqueda de soluciones relativamente pequeño, se optó por definir la siguiente vecindad:
Dada una solución parcial S, se saca un nodo v de dicha solución y se obtiene $S_1 = S \setminus \{v\}$, y se define un conjunto de candidatos a mejorar la solución parcial S. Sea C este conjunto de candidatos, los vértices pertenecientes a C son todos aquellos pertenecientes al grafo original que tienen al menos un vecino dentro de S_1. Una vez obtenido el conjunto de candidatos, se los va tomando en orden de grados (de mayor a menor) y se intenta insertarlos en S_1 siempre que esa inserción mantenga la propiedad de que S_1 sea una clique. Una vez hecho el intento con todos los vértices de C se verifica si se obtuvo o no una mejor solución que S, es decir, si $\#S_1 > \#S$. De ser positivo, esta solución S_1 es guardada.
Estos pasos se repiten para todos los vértices pertenecientes a S. Una vez finalizado esto, pueden suceder 2 cosas: que se haya encontrado una mejor solución que S o que no. Si se encontró una mejor solución, entonces se repiten todos los pasos antes mencionados pero para esta nueva solución parcial. Sino, significa que se encontró un máximo local, por lo que el algoritmo finaliza. Así como el orden en que se van tomando los vértices de C es de mayor a menor grado, en el caso de los vértices que se van descartando de S se hace en orden inverso, es decir, comenzando por los que tienen menor grado hacia los que tienen mayor grado.

Para graficar mejor este procedimiento, se detalla a continuación el pseudocódigo de las funciones que lo implementan. Recordar que dentro de los comentarios que contienen la complejidad, la letra "n" representa la cantidad de nodos de un grafo.

busquedaLocal(G: grafo)

```
1  termine = false;                                                    // O(1)
2  maxClique = cliqueConstructivo(G);                                  // O(n^3)
3  mientras termine == false hacer
4     cambio = false;                                                  // O(1)
5     <maxClique,cambio>= cambiarSiMaximiza(G,maxClique,cambio);
   // O(n^3)
6     si cambio == false;                                              // O(1)
7        termine = true;                                               // O(1)
8     fin si
9  fin mientras
10 devolver maxClique
```

Algoritmo 4: Pseudocódigo del algoritmo de búsqueda local

cambiarSiMaximiza(G: grafo, maxClique: conjunto, cambio: bool)

```
1  copiaClique = maxClique;                                            // O(n)
2  cliqueDeMenorAMayor = deCliqueAMinHeap(copiaClique);
   // O(n * log(n))
3  mientras cliqueDeMenorAMayor no sea un Heap vacío
4     posibleMejora = copiaClique \ {tope(cliqueDeMenorAMayor)};
   // O(log(n))
5     candidatos = vecinosAMaxHeap(posibleMejora);                     // O(n^2)
6     mientras candidatos no sea un Heap vacío
7        si vecinoDeTodos(tope(candidatos),posibleMejora);             // O(n)
8           posibleMejora = posibleMejora ∪ tope(candidatos);          // O(log(n))
9        fin si
10       pop(candidatos);                                              // O(log(n))
11    fin mientras
12    si #posibleMejora > #maxClique
13       maxClique = posibleMejora;                                    // O(n)
14       cambio = true;                                                // O(1)
15    fin si
16    pop(cliqueDeMenorAMayor);                                        // O(log(n))
17 fin mientras
18 devolver <maxClique,cambio>
```

Algoritmo 5: Pseudocódigo del algoritmo cambiarSiMaximiza

4.3. Análisis de la complejidad del algoritmo

Para analizar la complejidad del algoritmo de búsqueda local basta con comprender cómo se comporta tanto el algoritmo principal, llamado "busquedaLocal", como el secundario llamado "cambiarSiMaximiza". El resto de los algoritmos puede obviarse ya que, o bien ya fueron analizados en la sección [3] como es el caso de "vecinoDeTodos", o bien tienen un comportamiento trivial y conocido, como es el caso de las funciones "deCliqueAMinHeap" y "vecinosAMaxHeap" que como sus nombres lo indican, son funciones que devuelven un min-heap y un max-heap respectivamente. Lo que si es interesante aclarar de estas funciones es qué elementos devuelven en cada heap. En el caso de "deCliqueAMinHeap" toma todos los vértices que pertenecen a la clique que se le pasa como parámetro y los heapifica según sus

grados (de menor a mayor). En cambio, la función "vecinosAMaxHeap" toma todos aquellos vértices que no pertenecen a la clique pero que tienen algún vecino dentro de ella y los inserta en un max-heap ordenándolos por grado.

En primer lugar vamos a analizar al segundo algoritmo aquí presentado, es decir, "cambiarSiMaximiza". Como se observa dentro del pseudocódigo, existen dos ciclos dentro del algoritmo, uno dentro de otro. Para que el ciclo más global finalice, debe ejecutarse tantas veces como elementos haya en "cliqueDeMenorAMayor" por lo que a lo sumo se ejecutará n veces. Pasa lo mismo con el ciclo anidado dentro de este. El mismo se ejecutará tantas veces como vecinos de la clique haya, es decir, a lo sumo n. Dentro de este segundo ciclo, la función más costosa es la que se encuentra en la guarda del if que tiene una complejidad de $O(n)$. Como este ciclo se ejecuta a lo sumo n veces, la complejidad del mismo es $O(n^2)$. Ahora bien, además del ciclo anidado, la otra función costosa dentro del ciclo mayor es "vecinosAMaxHeap" que también cuesta $O(n^2)$. Por lo tanto, como el ciclo mayor se ejecuta a lo sumo n veces, la complejidad total de este algoritmo es $O(n^3)$.

Ahora bien, sólo faltaría analizar qué complejidad tiene la función principal. Pero si se observa bien, este algoritmo consta de un ciclo en el cuál existe una llamada a la función antes analizada. Por lo cuál sabiendo cuántas veces se ejecuta este ciclo, sabremos la complejidad total del algoritmo. Este ciclo finaliza cuando la variable "termine" valga $true$. La misma tomará este valor, cuando la variable "cambio" valga $false$ luego del llamado a "cambiarSiMaximiza". Por lo tanto, hay que volver al pseudocódigo de esta función para saber cómo se comporta allí esta variable.

La variable "cambio" sólo se modifica cuando existe una mejora en la clique mayor temporalmente hallada. Podemos ver entonces que el peor caso se da cuando la clique mayor temporal comienza con tamaño igual a 1 (respuesta obtenida con el algoritmo constructivo). Luego, dentro de cada llamada a la función "cambiarSiMaximiza", lo peor sería que sólo exista una mejora del tamaño en una unidad, es decir, en el primer llamado habrá un cambio de tamaño de 1 a 2, por lo que la variable "cambio" tomará el valor de verdad $true$. Cómo cada vez que se llama a esta función, esta mejora la respuesta temporal en una unidad, se puede observar que en el n-ésimo llamado el tamaño de la clique mayor temporal no aumentará, ya que se introdujeron todos los vértices existentes, por lo que en este llamado a la función secundaria, el algoritmo no entrará al if en el que se modifica a la variable "cambio", por lo que dicha variable quedará con valor $false$[4]. Por lo tanto, en el peor caso, en el n-ésimo llamado a esta función, la guarda del ciclo de la función principal se hace falsa. Entonces, como dicho ciclo se ejecutará a lo sumo n veces, y en el mismo hay un llamado a una función de complejidad $O(n^3)$, la complejidad total del algoritmo principal es $O(n^4)$.

[4]cabe notar que en cada llamado a la función "cambiarSiMaximiza" esta variable entra con valor $false$

5. Anexos